K. N. Toosi
University of Technology

Digital Image processing

CHW1

Due : March 18

Machine Vision & Image Processing Lab.

The Discrete Fourier Transform:

The FFT is a fast, O[$N\log N$] algorithm to compute the Discrete Fourier Transform (DFT), which naively is an O[N^2] computation. The DFT, like the more familiar continuous version of the Fourier transform, has a forward and inverse form which are defined as follows:

Forward Discrete Fourier Transform (DFT):

$$x_k = \sum_{n=0}^{N-1} x_n e^{-i\,2\pi\,kn/N}$$

Inverse Discrete Fourier Transform (IDFT):

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} x_k e^{i\,2\pi\,kn/N}$$

The transformation from $x_n \rightarrow x_k$ is a translation from configuration space to frequency space, and can be very useful in both exploring the power spectrum of a signal, and also for transforming certain problems for more efficient computation.

Because of the importance of the FFT in so many fields, Python contains many standard tools and wrappers to compute this. Both NumPy and SciPy have wrappers of the extremely well-tested FFTPACK library, found in the submodules numpy.fft and scipy.fftpack respectively. The fastest FFT I am aware of is in the FFTW package, which is also available in Python via the PyFFTW package.

Computing the Discrete Fourier Transform (25 point):

For simplicity, we'll concern our self only with the forward transform, as the inverse transform can be implemented in a very similar manner. Taking a look at the DFT expression above, we see that it is nothing more than a straightforward linear operation: a matrix-vector multiplication of $\vec{x}$,

$$\vec{X} = M\vec{x}$$

with the matrix M given by

$$M_{kn} = e^{-i\,2\pi\,kn/N}$$

With this in mind, compute the DFT using simple matrix multiplication as follows:

```
import numpy as np
def DFT_slow(x):
    """Compute the discrete Fourier Transform of the 1D array x"""
```

We can double-check the result by comparing to numpy's built-in FFT function:

```
x = np.random.random(1024)
np.allclose(DFT_slow(x), np.fft.fft(x))
```

```
True
```

Just to confirm the sluggishness of our algorithm, we can compare the execution times of these two approaches:

```
%timeit DFT_slow(x)
%timeit np.fft.fft(x)
```

```
10 loops, best of 3: 75.4 ms per loop
10000 loops, best of 3: 25.5 µs per loop
```

We are over 1000 times slower, which is to be expected for such a simplistic implementation. But that's not the worst of it. For an input vector of length N, the FFT algorithm scales as $O[N\log N]$, while our slow algorithm scales as $O[N^2]$. That means that for N=10^6elements, we'd expect the FFT to complete in somewhere around 50 ms, while our slow algorithm would take nearly 20 hours!

So how does the FFT accomplish this speedup? The answer lies in exploiting symmetry.

Symmetries in the Discrete Fourier Transform (25 point):

One of the most important tools in the belt of an algorithm-builder is to exploit symmetries of a problem. If you can show analytically that one piece of a problem is simply related to another, you can compute the subresult only once and save that computational cost. Cooley and Tukey used exactly this approach in deriving the FFT.

We'll start by asking what the value of X_{N+k} is. From our above expression:

$$X_{N+k} = \sum_{n=0}^{N-1} x_n . e^{-i\,2\pi\,(N+k)n/N}$$

$$= \sum_{n=0}^{N-1} x_n . e^{-i\,2\pi\,n} . e^{-i\,2\pi\,kn/N}$$

$$= \sum_{n=0}^{N-1} x_n . e^{-i\,2\pi\,kn/N}$$

where we've used the identity exp[2*π i n*]=1 which holds for any integer *n*.

The last line shows a nice symmetry property of the DFT:

$$X_{N+k} = X_k$$

By a simple extension,

$$X_{i.N+k} = X_k$$

for any integer *i*. As we'll see below, this symmetry can be exploited to compute the DFT much more quickly.

DFT to FFT: Exploiting Symmetry

Cooley and Tukey showed that it's possible to divide the DFT computation into two smaller parts. From the definition of the DFT we have:

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-i\,2\pi\,k\,n\,/\,N} \\ &= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i\,2\pi\,k\,(2m)\,/\,N} + \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i\,2\pi\,k\,(2m+1)\,/\,N} \\ &= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i\,2\pi\,k\,m\,/\,(N/2)} + e^{-i\,2\pi\,k\,/\,N} \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i\,2\pi\,k\,m\,/\,(N/2)} \end{aligned}$$

We've split the single Discrete Fourier transform into two terms which themselves look very similar to smaller Discrete Fourier Transforms, one on the odd-numbered values, and one on the even-numbered values. So far, however, we haven't saved any computational cycles. Each term consists of $(N/2)*N$ computations, for a total of N^2.

The trick comes in making use of symmetries in each of these terms. Because the range of k is $0\leq k<N$, while the range of n is $0\leq n<M\equiv N/2$, we see from the symmetry properties above that we need only perform half the computations for each sub-problem. Our $O[N^2]$ computation has become $O[M^2]$, with M half the size of N.

But there's no reason to stop there: as long as our smaller Fourier transforms have an even-valued M , we can reapply this divide-and-conquer approach, halving the computational cost each time, until our arrays are small enough that the strategy is no longer beneficial. In the asymptotic limit, this recursive approach scales as $O[N\log N]$.

This recursive algorithm can be implemented very quickly in Python, falling-back on our slow DFT code when the size of the sub-problem becomes suitably small:

```
def FFT(x):
    """A recursive implementation of the 1D Cooley-Tukey FFT"""
```

Here we'll do a quick check that our algorithm produces the correct result:

```
x = np.random.random(1024)
np.allclose(FFT(x), np.fft.fft(x))
```

```
True
```

And we'll time this algorithm against our slow version:

```
%timeit DFT_slow(x)
%timeit FFT(x)
%timeit np.fft.fft(x)
```

```
10 loops, best of 3: 77.6 ms per loop
100 loops, best of 3: 4.07 ms per loop
10000 loops, best of 3: 24.7 µs per loop
```

Our calculation is faster than the naive version by over an order of magnitude! What's more, our recursive algorithm is asymptotically $O[N\log N]$: we've implemented the Fast Fourier Transform.

Note that we still haven't come close to the speed of the built-in FFT algorithm in numpy, and this is to be expected. The FFTPACK algorithm behind numpy's fft is a Fortran implementation which has received years of tweaks and optimizations. Furthermore, our NumPy solution involves both Python-stack recursions and the allocation of many temporary arrays, which adds significant computation time.

A good strategy to speed up code when working with Python/NumPy is to vectorize repeated computations where possible. We can do this, and in the process remove our recursive function calls, and make our Python FFT even more efficient.

Vectorized Numpy Version (50 point):

Notice that in the above recursive FFT implementation, at the lowest recursion level we perform N / 32 identical matrix-vector products. The efficiency of our algorithm would benefit by computing these matrix-vector products all at once as a single matrix-matrix product. At each subsequent level of recursion, we also perform duplicate operations which can be vectorized. NumPy excels at this sort of operation, and we can make use of that fact to create this vectorized version of the Fast Fourier Transform:

```
def FFT_vectorized(x):
    """A vectorized, non-recursive version of the Cooley-Tukey FFT"""
```

Though the algorithm is a bit more opaque, it is simply a rearrangement of the operations used in the recursive version with one exception: we exploit a symmetry in the `factor` computation and construct only half of the array. Again, we'll confirm that our function yields the correct result:

```
x = np.random.random(1024)
np.allclose(FFT_vectorized(x), np.fft.fft(x))
```

```
True
```

Because our algorithms are becoming much more efficient, we can use a larger array to compare the timings, leaving out DFT_slow:

```
x = np.random.random(1024 * 16)
%timeit FFT(x)
%timeit FFT_vectorized(x)
%timeit np.fft.fft(x)
```

```
10 loops, best of 3: 72.8 ms per loop
100 loops, best of 3: 4.11 ms per loop
1000 loops, best of 3: 505 µs per loop
```

So how does FFTPACK attain this last bit of speedup?

"The pessimist sees difficulty in every opportunity. The optimist sees opportunity in every difficulty."